Looking in more detail @ what we did:

max $X_1 + 2X_2$

s.t.

$X_1 \le 20$ ← some mult. of this, say y_1

$X_2 \le 30$ ← " " " " y_2

$X_1 + X_2 \le 40$ ← " " " " y_3

$X_1, X_2 \ge 0$

positive mult. since o.w. $\le$ gets switched

add together

$$(y_1+y_3)X_1 + (y_2+y_3)X_2 \le 20y_1 + 30y_2 + 40y_3$$

could set $y_1+y_3=1$ and $y_2+y_3=2$

but actually, $y_1+y_3 \ge 1$ and $y_2+y_3 \ge 2$ works too since

$$X_1 + 2X_2 \le (y_1+y_3)X_1 + (y_2+y_3)X_2$$

when $y_1+y_3 \ge 1$ and $y_2+y_3 \ge 2$

want this upper bound to be as small as possible

Thus to find the best upper bound, need to solve the dual LP:

dual LP:

minimize $20y_1 + 30y_2 + 40y_3$

s.t. $y_1 + y_3 \ge 1$

$y_2 + y_3 \ge 2$

$y_1, y_2, y_3 \ge 0$

optimal soln $(y_1, y_2, y_3) = (0,1,1) \Rightarrow 20y_1 + 30y_2 + 40y_3 = 70$

Primal LP

maximize $X_1 + 2X_2$

s.t. $X_1 \le 20$

$X_2 \le 30$

$X_1 + X_2 \le 40$

$X_1, X_2 \ge 0$

optimal soln: $(X_1, X_2) = (10,30) \Rightarrow X_1 + 2X_2 = 70$

These agree, so we know the soln is optimal

Note: In converting the primal to the dual, we have:

1. replaced each variable w/ a constraint in the dual
2. replaced each constraint w/ a variable in the dual
3. replaced max w/ min in the dual

Theorem (weak duality)

A feasible soln to the dual LP is an upper bound on any feasible soln to the prime LP

Theorem (Strong duality)

The optimal solns to the dual LP and the prime LP agree.

Uses include: certificates of optimality

Shows finding feasible solns is at least as hard as finding optimal solns

Proving theorems (Konig's thm for max matching & min vertex cover)

Recasting LP's so they have fewer constraints/variables

Simplex in a nutshell

def Simplex

  let v be a vertex of the feasible region

  while $\exists$ vertex v', a neighbor of v with a better objective value

    set $v = v'$

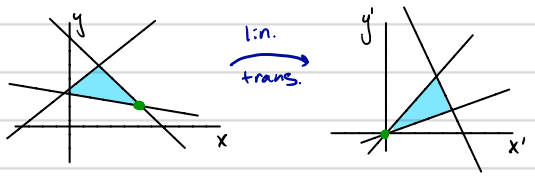
  return v

LP: $\max c^T x$ s.t. $Ax \le b$, $x \ge 0$

But! Its simplicity hides a lot of detail. Say LP has n variables and m inequalities

Vertex: a soln to a set of n inequalities (replacing inequality w/ equality)

neighbor: two vertices are neighbors if they share $n-1$ eqns in common



Wlog, say we start @ the origin. Since o.w. apply a linear transformation to get there

How do we find neighbors?

If all $c_i \le 0$, we are done (since incr. any x_i won't help, and all $x_i \ge 0$)

Say some $c_i > 0$, then increasing x_i will help. Note! Currently the ineq. $x_i \ge 0$ is tight (i.e. it's LHS = RHS). (Step a)

Release this tight constraint (i.e. ignore it)

Increase x_i until you run into another ineq. (i.e. some other ineq. becomes tight) ← this is actually done by solving a system of lin. eqns. $Bx = y$ to find by how much you need to increase x_i

You will now be at a new vertex. Record value of obj. func. and GOTO (Step a)

But how to find that initial vertex?

Recall alt. standard form: $\min c^T x$ s.t. $Ax \le b$, $x \ge 0$ (n vars., m inequalities)

Make RHS of ineq. all pos. (via algebraic manip.)

Create a new LP: introduce dummy vars $z = (z_1, \dots, z_m)$ and consider

$$\min z_1 + \cdots + z_m \quad \text{s.t.} \quad Ax + z = b, \quad x, z \ge 0$$

This has an obvious starting vertex! $x_1 = x_2 = \cdots = 0$, $z_1 = b_1$, $z_2 = b_2, \dots, z_m = b_m$

Run simplex on this LP and you'll get a soln $(x_1, \dots, x_m)$ and $(z_1, \dots, z_m)$ ← this will be a feasible starting vertex to your original problem

Note: ignoring degeneracy, cycling, etc.

Running time n vars, m constraints

each iteration: neighboor vertices share $n-1$ constraints $\Rightarrow \le n \cdot m$ neighbors

finding neighbors & computing value of obj. func turns out to be $O(nm)$ as well

How many iterations? $\le$ # vertices $= \binom{m+n}{n-1}$

$\therefore$ running time is $O\left(mn \binom{m+n}{n-1}\right)$ $\therefore$ exponential running time in n But! oddly, it's efficient in practice

eg Klee-Minty cube

$$\max 2^{n-1}x_1 + 2^{n-2}x_2 + \cdots + x_n$$

$$\text{s.t.} \quad x_1 \le 5$$
$$2^2 x_2 + x_1 \le 25$$
$$\vdots$$
$$2^n x_1 + \cdots + x_n \le 5^n$$
$$x_1, \dots, x_n \ge 0$$

$\Rightarrow$ simplex visits all 2^n vertices until final soln $(0, \dots, 5^n)$

But poly time algs exist! Narendra Karmarkar (grad student @ UC Berkeley) in 1984 came up w/ the first: the interior pt. method. Performs well in practice, but better on large problems (and not easy to warm start).

Complexity Classes, P, NP, and completeness

More of a survey (much more detail in CMPSC 464)

So far, we've seen algorithms that are $O(\log n)$ (binary tree search), $O(n^3)$ (Floyd-Warshall), and $O(2^n)$ (simplex)
We've also seen that some initialy naive algorithms can be improved to have better running time
eg Stupid sort (random perm, check if sorted) $O(n!)$
insertion sort $O(n^2)$
merge sort $O(n \log n)$

It would be nice to formalize what sorts of problems are inherently "easy" to solve (eg sorting) and which are "difficult" (eg longest path problem), and the relationships between them.

Disclaimer: Typically discussions re: complexity classes are predicated on the specific kind of algorithms under consideration eg decision problems: Those problems that have a yes/no answer (your textbook calls these "search problems")
optimization problems: require a value to be min/maxed
we will consider these
opt. problems can often be reduced to

So what is a complexity class?
Defn: A complexity class is the set of all problems for which there exists an algorithm that solves the problem in $O(f(n))$ where n = size of problem, f = fixed function

eg $TIME(n^k)$ = set of all problems that can be solved in $O(n^k)$ time
so all pairs shortest paths $\in TIME(n^3) \subseteq TIME(n^4)$
and sorting $\in TIME(n^2)$

This is enough to define our first important complexity class

even if we haven't invented/discovered such an algorithm yet

Defn $P = \bigcup_{k \in \mathbb{N}, \text{fixed}} TIME(n^k)$ i.e. all problems for which an algorithm exists that can solve it in polynomial time
"polynomial time"

Most of the problems we have seen are in P.
Now, $\exists$ many, many different complexity classes (over 500, probabilistic/randomized versions, space instead of time, parallelized versions, etc).
You would think the next to consider is:
Defn $EXP = \bigcup_{k \in \mathbb{N}} O(2^{n^k})$ exponential running time algs (eg simplex)

but instead, we focus on a class of problems that are easily verified to answer yes/no given a candidate soln, but for which the generation of soln's is TBD.

certificate/witness

Def'n: Let NP be the class of problems s.t. given an instance of the problem I and a proposed soln S, $\exists$ a polynomial time (decision) algorithm $C \in P$ (poly in $|I|$) s.t. $C(I,S) = \text{true}$ iff S is a soln to the problem. ("checker algorithm")

Note: Nothing is said about how hard/easy it is to generate candidate solns, no generate solns S s.t. $C(I,S) = \text{true}$

Note 2: "NP" stands for nondeterministic polynomial time (i.e. can be solved by a Turing machine that allows for probabilistic/nondeterministic transitions).

Thm $P \subseteq NP$

Proof: Recall that we are restricting our attention to decision problems (Yes/No T/F problems). Let T be a problem in P, wts $T \in NP$. Since $T \in P$, $\exists$ algorithm A s.t. for any instance I of T, $A(I)$ correctly decides if the instance I is possible (i.e. A decides I). Now given I and $S \in \{\text{True}, \text{False}\}$ let C be the algorithm:

```
def C(I,S):
    if A(I) = S
        return True
    else
        return False
```

Clearly, C has the same running time as A and so C is polynomial time, also $C(I,S) = \text{True}$ iff $A(I) = \text{True}$ iff S is a soln to I. □

Note 3: P = class of problems that can be decided in poly time

NP = " " " " " " checked " " "

So does there exist a problem $\in NP$ that is not in P? i.e. does $P \neq NP$? Or can all problems that can be checked in poly time also be solved in poly time? i.e. $P = NP$.

This is one of the biggest unsolved problems in CS!

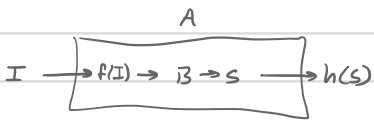
How would you even go after a problem like this? Idea: Go after the hardest problems in NP. Ones that we know of no efficient algorithm to solve. (but how do you define this?)

Using one of the most important tools in complexity class research: reductions

Def'n We say the problem A reduces to the problem B if $\exists$ poly. time algorithms f, h s.t. for I an instance of the problem A, $f(I)$ is an instance of the problem B, and for S the soln to $f(I)$, $h(S)$ is the soln of I. i.e.

$$I \longrightarrow f(I) \rightsquigarrow B \rightsquigarrow S \longrightarrow h(S)$$

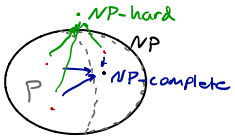
"$A(I)$"



In this case, write $A \rightarrow B$ (read "A reduces to B")

This means B is a "more powerful" alg. than A since B can solve any problem A can.

So to solve the question $P \stackrel{?}{=} NP$, we could find the "hardest" problems, and try to find poly. time solns to them (or show you can't). The def'n of "most powerful" is that of NP-complete.



Def'n A problem $A \in NP$ is *NP-complete* if all other problems in NP reduce to it.

Def'n A problem A (not nec. in NP) is *NP-hard* if all problems in NP reduce to it.

So we need to go after NP-complete problems

Note: If we know A is NP-complete and $A \to B$, then B is NP-complete too. Why? given any $C \in NP$, we know $C \to A$ and putting this together, we have $C \to A \to B$ so every problem C in NP reduces to B as well.

hence

Thm $P = NP$ iff $\exists$ a poly. time alg. to solve any NP-complete problem.

We have already seen some reductions: Bipartite matching → max flow → LP → ····· → (something NP complete)

The goal now is to find an NP-complete problem.

First, an example of a reduction

eg 3SAT → Independent set

3SAT: similar to horn formulas: Boolean variables, each clause has at most 3 pos/neg literals joined w/ "or", want to decide if $\exists$ variable assignment that makes all clauses true. eg $(\bar{x} \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{y})$, then $\bar{z}=T$, $y=T$, $\bar{x}=T$ works

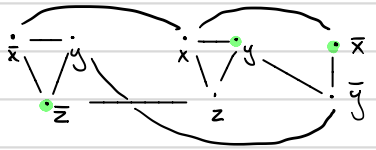
Independent set (IS): given $G=(V,E)$ and $t \in \mathbb{N}$, does there exist a subset of vertices $S \subseteq V$ w/ $|S|=t$ s.t. $\forall u,v \in S$, $(u,v) \notin E$? i.e. no two vertices in the subset adjacent. eg for $G =$  true for $t=1,2$ false for $t \geq 3$

Construct format conversion f: 3SAT instance → IS instance. Given an instance of 3SAT, let G be the graph:

1. vertices for each variable
2. edge between all variables in a clause
3. edge between pos/neg versions of the same variable
4. let t (# elem in indep. set) = # clauses

$(\bar{x} \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{y})$

$t=3$



This construction takes poly time

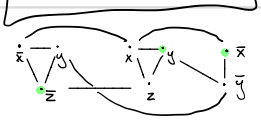
Then answer Indep. Set. question on this graph

If the answer is "yes" to ↑, then we can convert this to a soln of 3SAT: In indep set, set each selected variable to true, since each clause forms a clique in the constructed graph, we have a "true" in each clause. Valid assignment (i.e. $x=T$ and $\bar{x}=T$ doesn't happen) since we connected these vars. w/ edges.

If the answer to Indep set is "no", then want to show (no indep set) ⇒ (no 3SAT soln) or equiv (3SAT soln) ⇒ (indep set).

But this is easy: take the soln to 3SAT, it will form an indep set of size t by construction.

This is our second format conversion h: (soln of Indep set) → (soln of 3SAT)

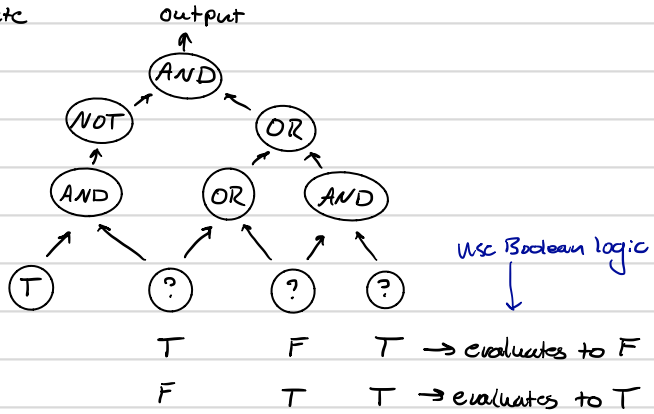
$(\bar{x} \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{y})$ →  → $\bar{z}=T$, $y=T$, $\bar{x}=T$

Note: It turns out, 3SAT is NP-complete (so Indep set $\xrightarrow{\text{reduces to}}$ 3SAT by def'n)
So combined w/ what we just showed (3SAT → Indep Set), this implies that the Indep. Set problem is NP-complete too.

Circuit SAT Goal will be to demonstrate this problem is NP-complete

Boolean Circuit: DAG whose vertices are one of 3 types:

1. AND or OR gates w/ in-degree 2
2. NOT gates w/ in-degree 1
3. Known input gates: in-degree 0, labeled T or F
4. unknown " " : " " ", labeled "?" can be assigned T or F

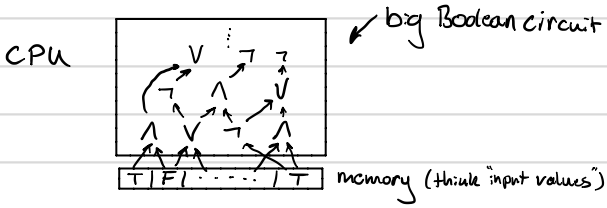


(CVP)

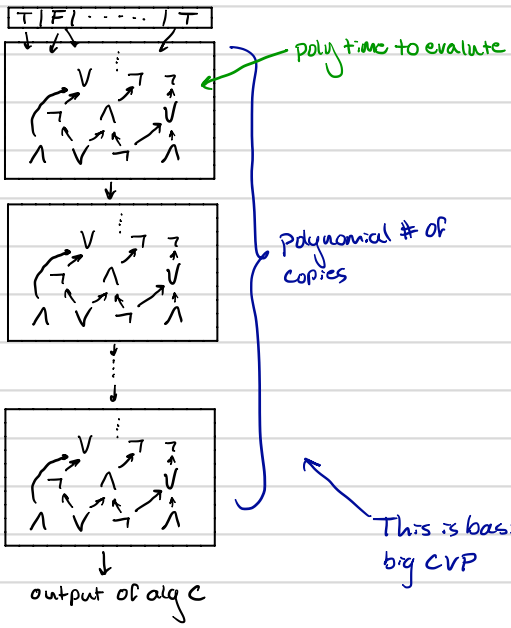
Circuit Value problem: Given Boolean Circuit w/ no unknown inputs, calculate value of output.

Clearly CVP ∈ P.

Important note! Say you have an algorithm C that runs in poly. time. Then we can implement it on a computer. But a computer is just a big circuit + memory. So running that algorithm is equivalent to running a polynomial number of CVP's.



So for each step of the algorithm C (and there are polynomial many steps), the algorithm is basically evaluating one Boolean circuit (each cycle on CPU)



and poly · poly = poly, so everything is still poly time.

So what we have actually shown is:

Thm Computing the result of any poly. time decision alg. C on some instance I reduced to an instance of the CVP.

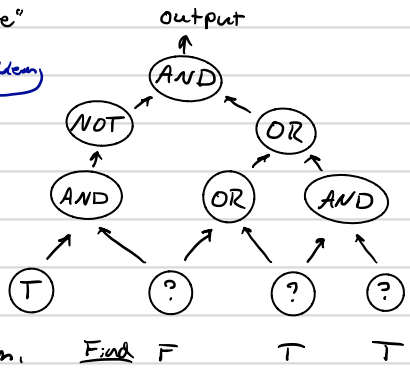
and hence:

Thm CVP is P-complete.

Proof: given any problem A ∈ P, ∃ poly time alg C to solve it on a given instance so by the above thm, we can directly convert this to the CVP. i.e. A → CVP.

Now consider:

Circuit Satisfiability (CSAT): given a Boolean circuit w/ unknown inputs, see if ∃ a satisfying assignment
i.e. if ∃ values for unknown inputs that make it evaluate to "True"



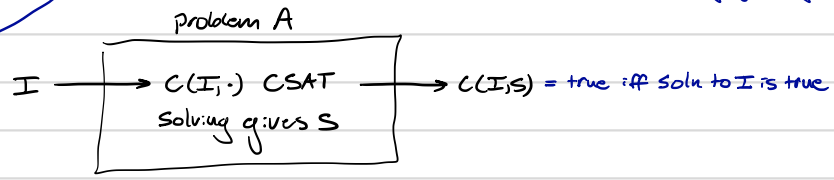
Thm: CSAT is NP-complete

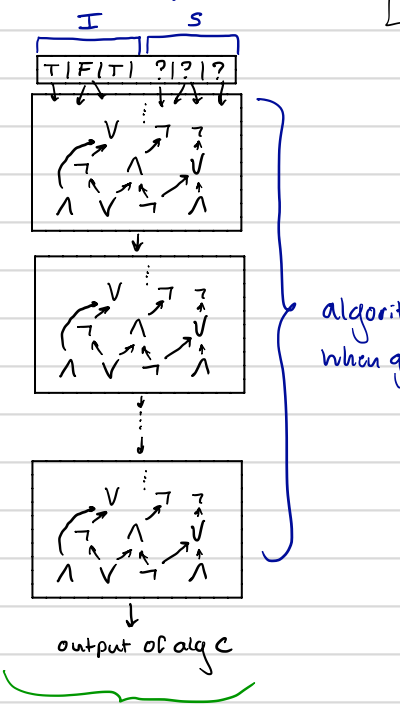
Proof: Clearly, CSAT ∈ NP since given a candidate soln, this is the same as the CVP (circuit value problem) which can be solved in poly time.

Now let A be any problem in NP. By def'n, ∃ poly time checking alg C s.t. for an instance I of A and a candidate soln S, C(I,S) = true iff the actual soln to I is true.

Using the same construction as before, we can thus cast C(I,·) as a CSAT problem. (see if ∃ a satisfying assignment)

So we now have





runs in poly time, so this big diagram has only polynomially many parts

algorithm for C(I,S) when given a candidate soln S

∴ A reduces to CSAT. □

Finding the values of "?" that make it evaluate to "true" is exactly the circuit Satisfiability problem CSAT

So long and thanks for all the fish!